University of Calabria

Department of Mathematics and Computer Science

Master Degree Course in Computer Science

Master Thesis

Multiclass Classification of Biosignals using several Supervised Learning approaches

in collaboration with

Faculty of Sciences of the University of Lisbon

IT-Instituto de Telecomunicações of Lisbon

Supervisors:
Prof. Francesco Ricca
Prof. Mário Calha

Candidate:
Davide Amato
ID 199670

Academic Year 2020/2021

Abstract

This work aims to solve a real problem faced by the research center "IT - Instituto de Telecomunicações" of Lisbon regarding one of its project, BITalino. BITalino is a modular hardware based on Arduino that has multiple input ports in which different sensors may be plugged-in in arbitrary order. These sensors are used for the biosignals monitoring (ECG, EDA, EEG, EGG, EMG, etc.).
The primary issue is that when a user plugs a sensor in some port, he should manually set the label related to that sensor by using the software OpenSignals. For example, if the user plugs the sensor for the ECG monitoring in the port number 2 then, through OpenSignals, he has to manually set that the sensor plugged in the port 2 is related with the ECG monitoring. One scope of this thesis is to automate this process.
Over the years, IT has accumulated a huge amount of data. Now I introduce the second problem that this work aims to solve.
In the process of saving biosignal data, because of the problem described above, BITalino keeps trace only of the relation *"data⟺input port"* but not of the relation *"data⟺sensor type"*. Hence, since any sensor can be plugged in any port, given a biosignal, if this hasn't been manually labeled by the user in the acquisition phase, it's hard to classify it. A way to classify a biosignal is visualizing it. But, to be able to classify it visually, you need a medical or specialized background. This work tries to automate the biosignals' classification phase so that it is possible to automatically classify both new and previously acquired data.
In this work several approaches are shown, all based on Supervised Learning. The algorithm used are: Support Vector Machines, K-Nearest Neighbors, Random Forest and Neural Networks.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 BITalino project

With the BITalino project, the research center "IT - Instituto de Telecomunicações" of Lisbon wants to offer a wide range of low-cost devices and open source hardware for education, rapid prototyping and exploratory research in biomedical engineering. BITalino's hardware is a modular hardware based on Arduino that has multiple channels in which different sensors for biosignal monitoring may be plugged-in. For real-time biosignals visualization, BITalino offers a software called OpenSignals, capable of direct interaction with BITalino. Among its core functionalities we can find sensor data acquisition from multiple channels and devices, data visualization and recording, as well as loading of pre-recorded signals. To know more about BITalino, see [6, 3]

1.1.1 Some application

BITalino has an undefinded number of possible applications. Over the years, the BITalino community has grown considerably and many projects have been made. To give an idea of how many different applications are possible with this hardware, you need only to think that BITalino has been used to create: a gesture-based drone controller, an high-end body monitoring garment that functions as a training system to avoid work related stress issues, a tangible interactive avatar displaying live physiological readings, a rehabilitation system for patients to reproduce a specific movement performed by the physiotherapist thanks to muscle sensing and FES and many other projects.

1.2 Motivation

This work aims to solve some problem regarding the BITalino project.

The primary issue is related to the fact that this hardware allows the user to connect different sensor combinations that have to be configured according to his individual set up. During the usage of BITalino the user, through the software OpanSignals, can visualize in an easy way the data acquired by the sensors. For each channel that the device has, there is a label realtive to the channel (CH1,...,CHn) and a label relative to the sensor type associated to the channel (A1,...,An). For instance, if the hardware has 6 channels, the labels relative to the channels will be CH1,...,CH6 and the labels relative to the sensor types will be A1,...,A6. To better understand and recognize the signals, when a user plugs a new sensor he should manually set the label of the sensor type relative to the channel. For example, if a user plugs an ECG sensor into the channel 'CH3', he should manually change the sensor type's label of the channel 'CH3' from 'A3' to "ECG". Sometime the user doesn't know which type of sensor he is plugging and, without medical or specialized background, he's not able to recognize it by visualizing it. With this work. I will automate this labeling process so that when the user will plug some sensor, OpenSignals will be able to automatically set the right sensor type's label of the relative channel.

The second issue is strongly related to the first. Over the years, the company has accumulated a huge amount of data. Because of the problem described above, many BITalino's datasets keeps trace of the relation *"data⟺channel"* but not of the relation *"data⟺real sensor type"*. Hence, since any sensor can be plugged in any port, given a biosignal, if this hasn't been manually labeled by the user in the acquisition phase, it's hard to classify it as ECG, EMG, etc... A way to classify a biosignal is visualizing it. But to be able to classify it, the user needs a medical or specialized background.

This thesis tries to automate the biosignals' classification phase so that it is possible to classify all the data previously acquired.

1.3 Proposal

The proposal presented here consinsts of several supervised learning approaches for the classification of different biosignals. The signals covered by this work are ECG, EDA, EMG and RESP. The algorithms used in this work to solve this task are: Support Vector Machines (SVM), K Nearest Neighbours (KNN), Random Forest (RF), Dense and Convolutional Neural Networks (DNN and CNN respectively). All of them, including their imple-

mentation, are discussed in **chapter 4 (Data Preprocessing)**

1.4 Document structure

This document is structured in the following way:

- **chapter 2 - Background knowledge**: some background information needed for a better understanding of the entire work. It will introduce the biosignals treated and several Machine Learning related concepts.
- **chapter 3 - Problem statement**: a detailed description about the problems that this work amis to solve.
- **chapter 4 - Data Preprocessing**: it contains an overview of the preprocessing flow implemented to prepare the data for the supervised learing algorithms and a detailed description for each step involved.
- **chapter 5 - Solutions**: the solutions proposed within this work, giving all the insights and details on the implementation of the ML algorithms used.
- **chapter 6 - Conclusion and future works**: the last chapter that summarizes the results obtained with the solutions implemented and gives an overview on what could be done in future works in order to improve this work

Chapter 2

Background knowledge

In this Chapter you can find a description of all type of biosignals that can be acquired by BITalino hardware, some notion about machine learning, the classification problem, an overview about the supervised and unsupervised learning approaches used in this work, recurrence plots and a brief list of the technologies used.

2.1 Biosignals

A **biosignal** is nothing more than a signal of an organism which changes and, therefore, can be measured and monitored over time. A biosignal can be either a *bioelectrical signals*, that can be measured through changes in electrical potential across a specialized tissue, organ or cell system like the nervous system, or *non-electrical signals*.
The biosignals taken into account are the following:

- **ECG**: it's an electrogram of the heart which is a graph of the small electrical changes over time that are a consequence of cardiac muscle relaxation (diastole) and contraction (systole) during each cardiac cycle (heartbeat).
- **EMG**: it detects the electric potential generated by muscle cells when these cells are electrically or neurologically activated
- **EDA**: it measures the continuous variation in the electrical characteristics of the skin.
- **RESP**: by using a chest-belt, it measures changes due to chest expansion during each respiration cycles.

2.2 Machine Learning

Machine learning (ML) is a science field belonging to Artificial Intelligence (AI) that aims to study algorithms capable of learning from experience and from (generally huge amount of) data. It collects several methods including: artificial neural networks, pattern recognition, image processing, data mining and many others. ML algorithms create models able to make predictions or decisions without being explicitly programmed to do so, thanks to the knowledge previously acquired in the learning (training) phase.

2.2.1 Classification problem

Classification is one of the most common tasks in decision making. A classification problem consists to assign a class to an observation (or observations). This assignment is done by finding in the observation some characteristics or rules, learned by the classification algorithm in the training phase, that identify the belonging to a particular class.

2.2.2 Supervised and Unsupervised Learning

In ML there exist various types of learning processes. Two of the most popular are **Supervised** and **Unsupervised learning** (**SL** and **UL**). Both SL and UL algorithms are able to predict future instances by recognizing general patterns and characteristics of the domain of interest. The key factor that differentiates SL from UL algorithms is that while UL algorithms learn these patterns and characteristics from **unlabeled** and **unknown data**, SL algorithms are trained by using **labeled instances** and produce an inferred function able to map *unseen* instances.
Supervised learning is well known for its effectiveness to solve classification problems and for this reason it has been chosen as learning process for this work

2.2.3 Generation of Train, Validation and Test Sets

As it has already been stated, supervised ML algorithms base their ability to solve problems on the learning that take place during a training phase. Durign this phase these algorithms learn from known instances in order to draw conclusions about new unseen instances.
The *modus operandi* in supervised learning is to split the whole dataset into 3 separeted sets: **Training**, **Validation** and **Test sets**.

Training set This is the set used by the algorithm during the training phase in order to capture all the features they need to solve new instances. It contains a set of both examples and solutions of the problem that the algorithm sees to **fit model's paramenters** (internal variables such as weights of connections in neurons of a NN).

Validation set It is used to have an unbiased estimate the ability of the model to predict new instances. Unlike the training set, it is not directly used to update model's parameter but to have an estimate of the model's error rate on the test set while **tuning the hyperparameters** (external variables such as the number of neurons per layer, number of hidden layers, etc.). There are other ways to evaluate the model. One of the most popular is the *Cross-validation* method described below.

Test set It contains examples never seen before by the model during the training phase. It is used when model performance on the validation set is satistying in order to get an **unbiased evaluation of the final model**.

2.2.4 Cross-Validation

Cross-Validation (**CV**) is a method used to validate the model during training and to tune model hyperparameters. Hence, when CV is used, the creation of the validation set is no longer needed. It is an iterative process that consists of splitting the data in training and validation sets to train and validate the model within a certain number of iterations.There are several CV implementations. One of the most popular is the **K-Fold CV**

K-Fold Cross-Validation It's the simplest implementation of the CV. There is a parameter called k that has to be chosen and refers to both:

- the number of portions into which the training set must be splitted
- the number of iterations the procedure must perform

Generally there is an initial random shuffle of the dataset to ensure that data is distributed uniformly and to avoid that too many samples with the same class label are contiguous. Later, the shuffled dataset is splitted into k subsets that are complementary to each other (they do not share any element). In each iteration a subset is taken as validation set, the rest of the dataset is

used to fit the model and the score obtained by the model on the validation set is stored. It's important to notice that each subset is taken as validation set only once. In the end an evaluation of the scores saved during the training phase is done in order to make a final judgement on the model skills.
Very common is the 10-Fold CV, where the parameter k is equal to 10.

Algorithm 1 K-Fold Cross-Validation procedure

Require: dataset $S = (x_1,\ y_1), ..., (x_n,\ y_n),\ k > 1$
  $S = randomShuffle(S)$
  $groups = split(S,\ k)$ //split S into k complementary groups
  $scores = [\ ]$
  for all g **in** $groups$ **do**
    $valid\_set = g$
    $train\_set = groups/g$
    $model = generateLearningModel()$
    $model.fit(train\_set)$
    $score = model.evaluate(valid\_set)$
    $scores.add(score)$
  end for
  $evaluate\_performance(scores)$

2.2.5 ML Algorithms

2.2.5.1 Support Vector Machine

Support Vector Machine (**SVM**) is a very robust supervised ML algorithm that performs very effectively for both classification and regression task. SVM will place every data point in an n-dimensional space (where n is the number of features), using the value of each feature of a certain data point as its coordinates. Hence, it tries to find hyper-planes capable to separate the various classes.
SVM is based on the concept of Support Vectors. They are nothing else but coordinates of data points that *"supports"* more than others the discovery of the hyper-planes that better separate the classes

2.2.5.2 K-Nearest Neighbors

KNN is the simplest ML algorithm for classification tasks. It simply examinates the *k-neighbors* of each data point and calculate the distance betweem their feature vectors. Different distance methods could be chosen (Euclidean

Distance, Manhattan Distance, Minkowski Distance to name a few). It classify a certain instance basing its decision on the most recurring class among the k closest examples in the training data.

2.2.5.3 Random Forest

This method, like those described above, can be used for tasks concerning both classification and regression. It is based on the concept of **Decision Tree** (**DT**).
DTs are a very common predictive model in statistics. In this model leaves represents the target value (that can be classes in classification tasks, or real numbers in regression tasks), while the branches represents conjuctions of features. Studing the various examples in training phase (hence, the path in the DT for each training example that leads to the relative class or DT's leaf), a DT is able to observe a new instance and to draw conclusions about its class.
The **Random Forest** (**RF**) operation consists in creating many decision trees during the training phase. In classification contexts, the output of the **RF** is decided by a majority-voting among the outputs of the internal decision trees.

2.2.5.4 Neural Networks

Neural Networks (**NNs**) are part of the huge family of ML methods. In particular, they are the core of Deep Learning algorithms. NNs are computing systems that tries to emulate the way that biological neurons works in the human brain.
NNs can be seen as oriented graphs. The smallest part of a NN, i.e the counterpart of a biological neuron, is a mathematical model called **perceptron** (or **artificial neuron**). A perceptron is a node that has some **inputs**, **weights**, a **threshold** (**bias**) and an **output**. If the output produced by the node is above the given threshold, the node is *activated* and propagate the signal to the next layer of the network. The values of weights and biases of perceptrons are self-updated in the training phase through **gradient descent**, based on the loss function chosen, in order to minimize errors.
In a NN several perceptrons are stocked together to form a **layer**. Then, different layers are placed one after the other to form a NN. The structure of a NN is the following:

1. **Input Layer**
2. **Hidden Layer(s)**

3. **Output Layer**

When there are more than one Hidden layer, we talk about Deep Neural Networks. From here on I will use NN to refer to Neural Network in general, including deep ones.

Dense Neural Network (DNN) Also known as **Fully-Connected NN** (**FCNN**), it is the simplest type of NN. In **DNNs**, every node of a layer is connected to every node of the next layer. These NNs are very flexible models, adaptable to a huge range of problems. Generally it's easy and fast to implement a DNN that is able to achieve good results on a problem but, as it is easy to understand, it tend to have worst performances compared to NNs specifically desined for a certain kind of problem.

Convolutional Neural Network (CNN) This type of NN is explicitly designed to handle images as inputs. A tipical CNN architecture involves mainly 3 kind of layers:

1. **Convolutional Layers (Conv layers):** they are used to extract feature map from the input by applying a mathematical operation called *convolution*. Convolution is a linear operation that performs a dot product between the input data (tipically a multidimensional matrix) and a smaller matrix called kernel (or filter). The kernel moves from left to right, top to bottom of the input to explore and perform the dot product to all the filter-sized part of the image.
   Using multiple Conv layers we are able to extract features of increasing complexity. The first layer acts directly to the original pixels and extract low level features, such as lines. The second layer acts on the feature map outputted by the first and it can extract features (such as triangles, squares) resulting from the combination of the low-level features extracted in the first layer, and so on.

2. **Pooling Layers (Pool layers):** these layers come after the convolutional one and is used to downsample the data (i.e. dimensionality reduction). Hence, it's used to make the NN less computationally heavy and, at the same time, to extract dominant features. Like the convolution layer, it performs its task by moving a kernel on each element of the feature map and performing an operation that depends on which type of pooling method (Maximum, Minimum, Avarage or Adaptive Pooling) we choose.

3. **Fully-Connected Layers (FC Layers):** they are tipically placed right before the output nodes and are used to learn non-linear combinations of the high-level features taken as input by the Conv layer. Before feeding the feature map to the FC layer, this is flattened into a 1D array

2.3 Technology used

The programming language used to implement the solution presented in this work is **python**, well known for its foucs on readability and extremely popular nowadays in ML contexts, along with **Keras**, an open-source library that acts as an interface for the **TensorFlow** library, which is one of the most famous library for AI and ML tasks.

Chapter 3

Problem statement

This work wants to offer to the research center IT a tool to automatize the process of classification of a biosignal. To better understand the problem without getting too technical, I'll present a simple overview on the BITalino board [Fig.3.1]. BITalino is a modular hardware based on Arduino. It is composed by a main board and multiple input ports (channels) in which different sensors for biosignal monitoring may be plugged in different order. And that is exactly where the problems arise.

First issue The first issue is that when a user plugs a sensor in some input port, the software used to manage settings and visualize data, called OpenSignals, doesn't know which kind of sensor has been plugged in that channel. That said, it's easy to imagine that it's hard for the user to figure out which kind of signal he is looking at if he forget the configuration done before. To make an example, suppose the user plugged the sensors to measure ECG, EDA, EMG, temperature, acceleration (using a 3 channels sensor) and respiration in A1,...,A6 channelsbut he doesn't remember precisely the

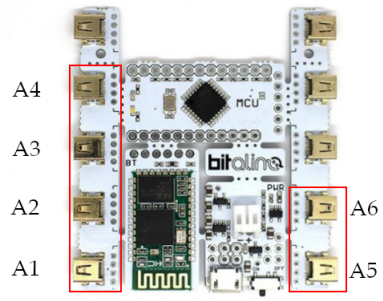


Figure 3.1: BITalino Board with connection ports for sensors and actuators. Sensors for connection to an electrode cable marked in red (A1-A6)

configuration. In this case it's hard for the user to understand without any help which sensor is plugged in which port.
To avoid this problem, when a user plugs a sensor in some channel, he, through OpenSignals, should manually set the label related to that sensor. Doing this, OpenSignal can keep trace of the relation between "datastream" and "sensor type". But, very often, users forget to change the label of the sensor.

Second issue The second issue is strongly related to the first. If a user doesn't tell OpenSignal the type of sensor plugged in the relative channel, when he saves the monitoring session to file, inside the file (usually it's in hdf5 format) the sensor's type is unknown. In this way, if someone read this file is not able to figure out which type of sensor was plagged in which port. A way to overcome this problem could be to classify the biosignal by visualizing it. But this is not a very suitable solution. First of all, to be able to classify it, the user needs a medical or specialized background. Second, this solution can be applied if you are dealing with a limited number of files. If there are dozens, hundreds or even thousands of files to classify, this is not a practicable solution anymore.

Chapter 4

Data Preprocessing

Data preprocessing is a crucial step in Machine Learning. The ability of a model to learn is strictly related to the quality of the data given to it. Therefore, it is clear that we have to preprocess our data before feeding to ML algorithms. To solve the classification problem described above, three different approaches have been tested (described in detail in chapter 5). Nevertheless, most of the preprocessing steps are common to all of them (steps 1. to 6. in Fig. 4.1), exect those specific to certain approach (7.a and 7.b in Fig. 4.1).

4.1 Dataset description

Firsly, I want to briefly describe the dataset. The dataset used for this work is the **WESAD dataset** [20]. It has been used in many other works including [16][19][7] and contains physiological and motion data of 15 subjects recorded during a lab study. WESAD data has been recorded using two device: a chest-worn device (RespiBAN) and a wrist-worn device (Empatica E4). The following sensor modalities are included: blood volume pulse, electrocardiogram (ECG), electrodermal activity (EDA), electromyogram (EMG), respiration (RESP), body temperature, and three-axis acceleration, with a sampling rate equal to 700. Moreover, the dataset take into account stress and emotions, by containing three different affective states (neutral, stress, amusement). In addition, self-reports of the subjects, which were obtained using several established questionnaires, are contained in the dataset.
The only data retrieved and used for this work is the one recorded with RespiBAN device and relative to ECG, EMG, EDA and RESP data.

Note: Unfortunately, due to hardware and time limitations, from the data of

each subject I used only the first million rows. So, for each patient, 1 million samples of each signal (ECG, EMG, EDA, RESP) have been used. Considering the sampling rate of 700, they correspond to the first 24 minutes of recording.

4.2 Preprocessing steps

4.2.1 Transform raw data in standard unit

WESAD raw data values are not given in a standard form. Hence, to convert these values into standard units of measure I used the tranfer functions written in the BITalino's datasheets.
Listed below you'll find the equations used to convert the data in standard units, where *signal* contains *raw sensor values*, $vcc = 3$ and $chan\_bit = 2^{16}$:

- EMG (mV): $((signal/chan\_bit - 0.5) * vcc)$
  Details: `https://bitalino.com/storage/uploads/media/revolution-emg-sensor-datasheet-1.pdf`
- ECG (mV): $((signal/chan\_bit - 0.5) * vcc)$
  Details: `https://bitalino.com/storage/uploads/media/revolution-ecg-sensor-datasheet-revb-1.pdf`
- EDA (μ S): $(((signal/chan\_bit) * vcc)/0.12)$
  Details: `https://bitalino.com/storage/uploads/media/eda-sensor-datasheet-revb.pdf`
- RESPIRATION (%): $(signal/chan\_bit - 0.5) * 100$
  Details: `https://bitalino.com/storage/uploads/media/pzt-sensor-datasheet-revb.pdf`

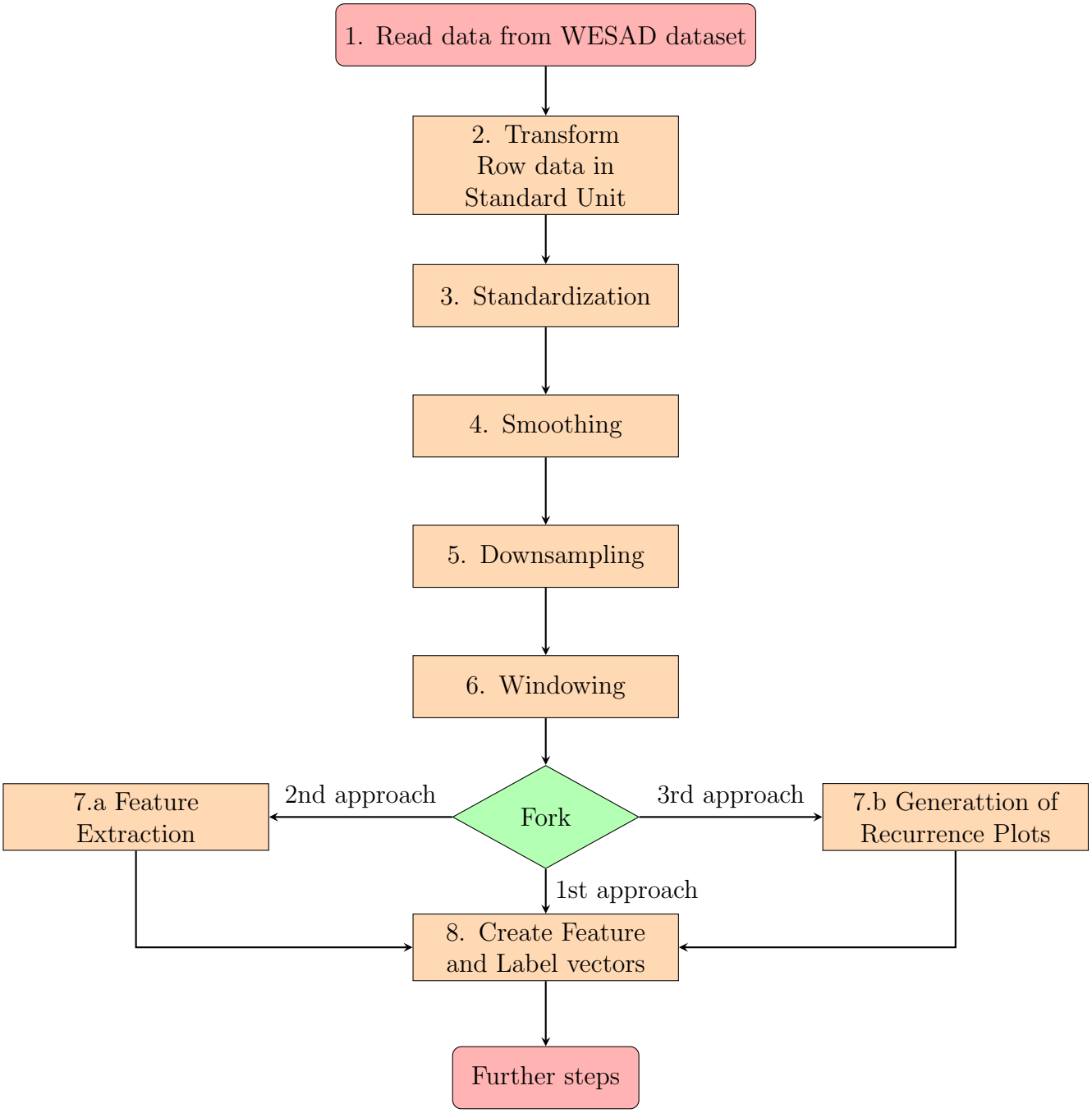
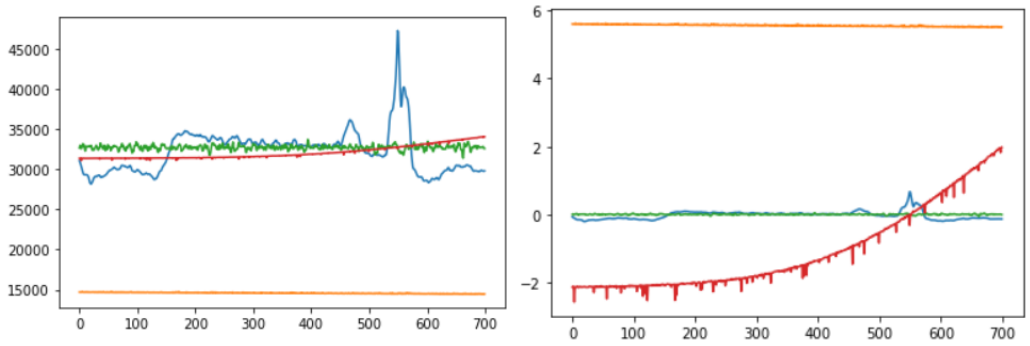


Figure 4.1: Diagram of the preprocessing steps



(a) Before trasformation

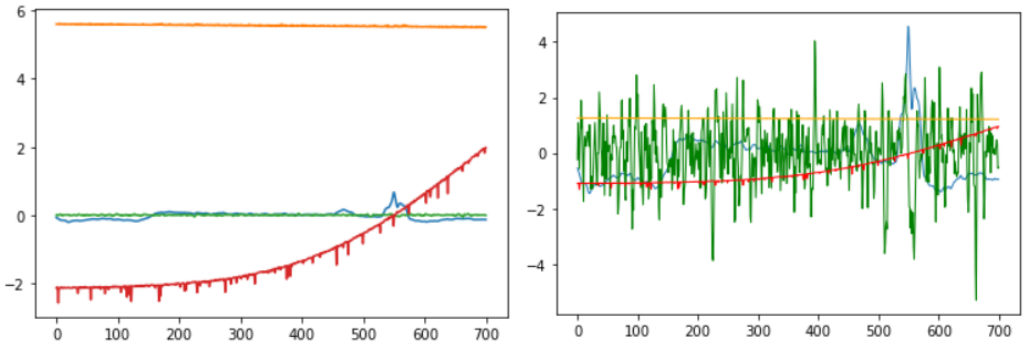
(b) After trasformation

Figure 4.2: Plots of the first 700 samples of 2nd Patient's signals before (a) and after (b) raw data trasformation in standard unit (ECG in blue, EMG in green, EDA in orange and RESP in red)

4.2.2 Standardization

In ML, a technique often applied as part of data preparation is the **Standardization**. It is broadly used in Time Series Classification (TSC) [2] The goal of standardization is to change the values of numeric columns in the dataset to use a common scale, without distorting differences in the ranges of values or losing information. Standardization is also required for some algorithms to model the data correctly.
This technique is used to center data around the mean value with a unit standard deviation. In this way, after the standardization process the resulting data will have a **mean** of **0** and a **standard deviation** of **1**.
The mothod used to standardize the data is the popular **Z-score normalization**, that is computed with the following formula:

$$x_{stand} = \frac{x - mean(x)}{standard\ deviation(x)}$$



(a) Before standardization

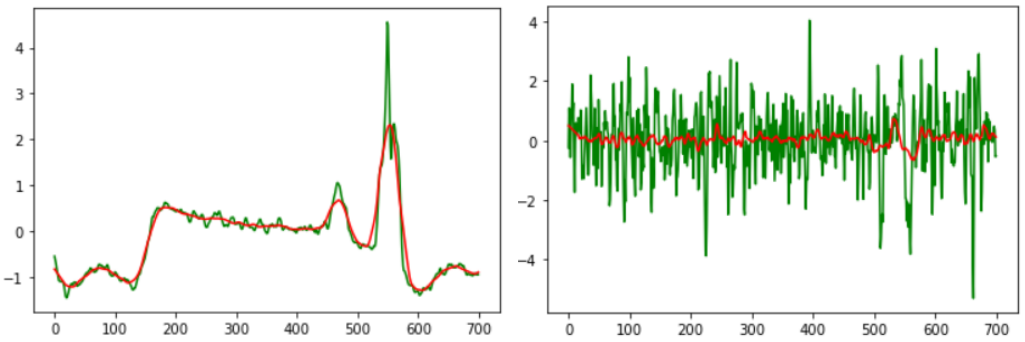
(b) After standardization

Figure 4.3: Plots of the first 700 samples of 2nd Patient's signals before (a) and after (b) standardization (ECG in blue, EMG in green, EDA in orange and RESP in red)

4.2.3 Smoothing

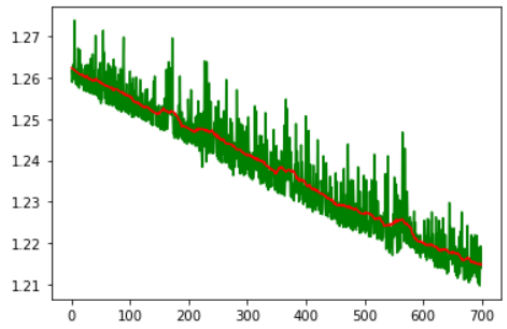
To smooth signals data the **Savitzky–Golay (savgol) filter** [18] has been applied. It is a digital filter commonly used as a preprocessing step in signal processing. It uses data points for smoothing the graph.
For a given signal measured at N points and a filter of width w, savgol calculates a polynomial fit of order o in each filter window as the filter is moved across the signal [8].
For each patient's data I stored the first 1 million rows the savgol filter has been applied to all signals and has been paramiterized using the following values:

- number of points: 1 Million
- window's size: 31
- polynomial order: 1

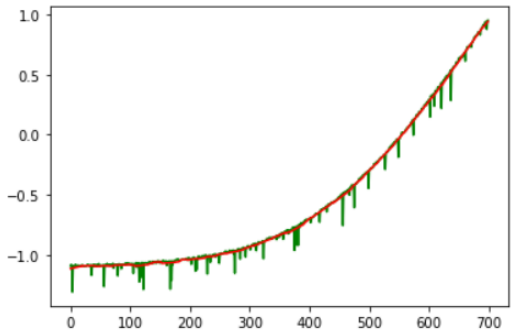


(a) ECG smoothing

(b) EMG smoothing

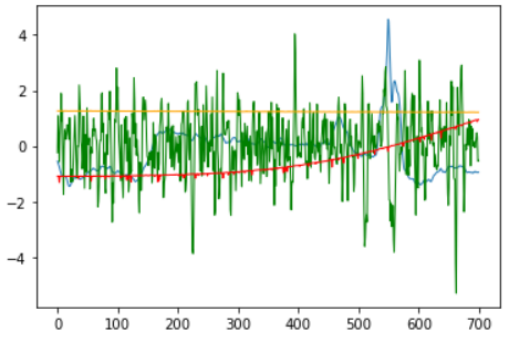


(c) EDA smoothing

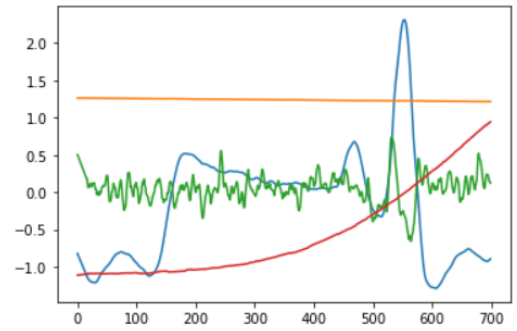


(d) RESP smoothing

Figure 4.4: Plots showing the effect of smoothing on the various signals (original signal in green, smoothed one in red)



(a) Before smoothing



(b) After smoothing

Figure 4.5: Plots of the first 700 samples of 2nd Patient's signals before (a) and after (b) smoothing (ECG in blue, EMG in green, EDA in orange and RESP in red)

4.2.4 Downsampling

To reduce the amount of data to work with, I choose to decrease the sample rate from a higher value of 700 to a lower value of 100. Therefore, data has been decreased by 7 times without any relevant loss of information.

4.2.5 Windowing

This is one of the mostly used processes in digital signal processing. **Windowing** allows to deal with very large datasets following the *divide et impera* paradigm. This process consists into divideing the original dataset into small subsets that can be easly processed and analyzed one by one.
There are many windowing functions that can be used (Hamming Window, Sum-Cosine Window, Blackman Window, etc...). In this work I used a naive approach that consists in a simple division of the dataset in subsets of lenght 250. Considering that now, after the downsampling step, the sample rate is equal to 100 (100 samples in 1 second), each window will results in a 2.5 seconds signal.

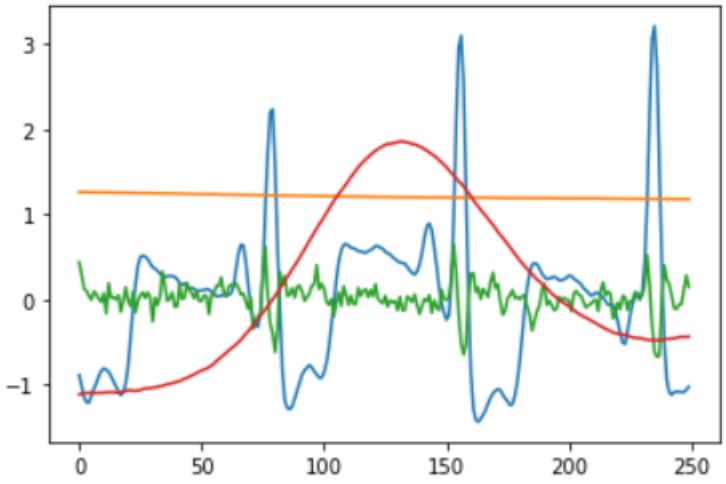


Figure 4.6: Plots showing a superimposed view of the first 2.5 seconds windows of 2nd Patient's signals (ECG in blue, EDA in yellow, EMG in green, RIP in red)

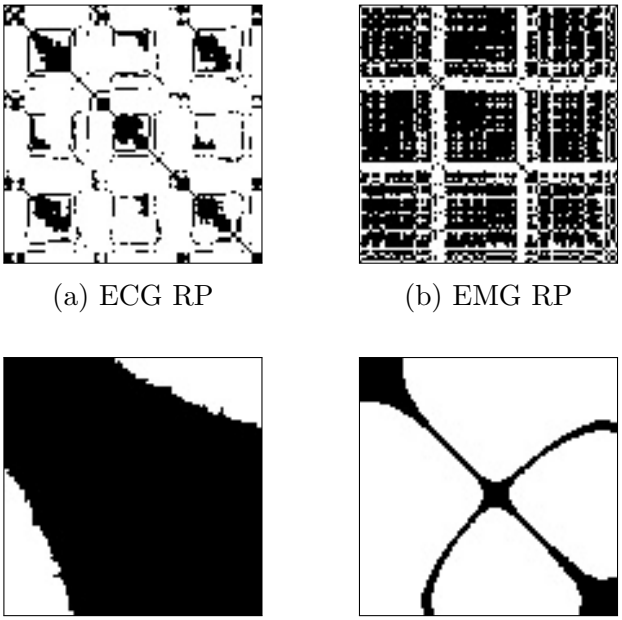
4.2.6 Feature Extraction

Feature Extraction (**FE**) is used very often during preprocessing for ML algorithms. It consists in creating, from an initial dataset, new features

that are able to represent the central properties of the dataset in a lower-dimensional space to facilitates the learning process [15]. FE can be used to extract features accross different domains like Temporal, Statistical, Spectral, Non-Linear. For this work I extracted features, belonging to the Statistical Domain, commonly used in other works concerning the application of ML algorithms in the field of physiological signals [5]. In particular, the following features have been extracted: **mean**, **standard deviation**, **kurtosis** and **skewness**.

4.2.7 Generation of Recurrence Plots

A **Recurrence plot** (**RP**) is a powerful tool in time series analysis.
Natural processes can have a distinct recurrent behaviour, e.g. periodicities (as seasonal or Milankovich cycles), but also irregular cyclicities (as El Niño Southern Oscillation). Moreover, the recurrence of states, in the meaning that states are arbitrary close after some time, is a fundamental property of deterministic dynamical systems and is typical for nonlinear or chaotic systems. The recurrence of states in nature has been known for a long time and has also been discussed in early publications (e.g. recurrence phenomena in cosmic-ray intensity, Monk, 1939).
The computation of a RP will result a square matrix where every element corresponds to those times at which a state of a dynamical system recurs. Techniqually, the RP reveals all the times when the phase space trajectory of the dynamical system visits roughly the same area in the phase space [1]. The implementation of the RP algorithm used is available on [13]. There are 3 parameters requested by this implementation of the RP function: the *data* (that is mandatory), the *eps* (epsilon) parameter (default: 0.1) and the *steps* parameter (default: 10). To generate RP for this work I used the 2.5 sec windows produced in the *windowing* step of the preprocessing as first parameter and an *eps* value 0.025, using the default *steps* value.
In the end, the generated RPs have been compressed from an original shape of 250x250 to a smallest shape of 100x100 using a bicubic interpolation over 4x4 pixel neighborhood. Resulting RPs can be seen below [Fig.4.7]



(a) ECG RP (b) EMG RP

(c) EDA RP (d) RESP RP

Figure 4.7: RPs of the signals belonging to the first window of the 2nd Patient [Fig.4.6]

4.3 Generation of Feature and Label Vectors

Before feeding data to a ML algorithm, we need to reshape it. There have been used several methods to create feature and label vectors. The choise of the method used depends on the approach chosen. In this section I will show all the type of reshaping implemented for this work.

4.3.1 Creation of the Feature Vector

4.3.1.1 Windows Feature Vector

It's simply generated by concatenating the windows (250 samples each, equivalent to 2.5 seconds) obtained during the step **4.2.5 Windowing** (7th step of the workflow [Fig. 4.1]).

	1	2	3	...	250
window 1	-0.88366709	-1.01818296	-1.14511049	...	-1.02084599
window 2	-1.02071844	-1.04687701	-1.13539463	...	-1.41214319
window 3	-1.61578489	-1.63817194	-1.59459076	...	-1.04664654
...	...	...	...	...	...
window N	0.21226119	0.2214532	0.23046906	...	5.37687263

Figure 4.8: Representation of the Windows Feature Vector

4.3.1.2 Statistical Feature Vector

This feature vector is generated by extracting from each window the selected statistical features (**4.2.6 Feature Extraction**).

	Mean	Std Dev	Kurtosis	Skewness
window 1	-0.0064701	0.86148317	1.99310218	0.88430783
window 2	0.01353117	0.9205743	1.45839007	0.75916071
window 3	-0.04111563	0.89925567	1.38411502	0.56375127
...	...	...	...	...
window N	1.08263866	1.36027982	2.76093172	2.04699039

Figure 4.9: Representation of the Statistical Feature Vector

4.3.1.3 Images Feature Vector

This is the only feature vector used that contains 2D features. It is created by stacking together all the recurrence plots generated in **4.2.7 Generation of Recurrence Plots**.

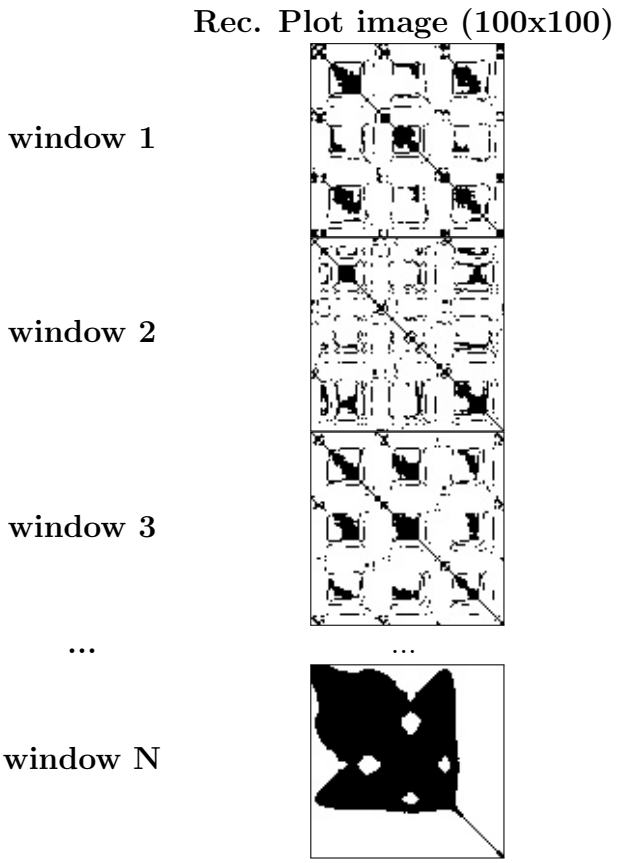


Figure 4.10: Representation of the Image Feature Vector

4.3.2 Creation of the Label Vector

4.3.2.1 Plain Label Vector

This vector is made by the label correponding to each window in the feature vector. The labels used are 'ECG', 'EMG', 'EDA' and 'RESP' for the respective signal.

	Plain label
window 1	"ECG"
window 2	"ECG"
window 3	"ECG"
...	...
window N	"RESP"

Figure 4.11: Representation of the Plain Label Vector

4.3.2.2 One-Hot Encoded Label Vector

In ML contexts, the **one-hot encoding** consists to map each element in a set with a group of bits among which there is a single bit equal to 1 (one-hot) and all the others equal to 0, contraposed to the **one-cold encoding** where there is only one bit equal to 0 and all the others equal to 1. The number of bits needed to create the mapping is equal to the number of elements to represent.
Since in this project I have 4 labels to map, I used a 4 bits on-hot encoding, mapping the labels as follow:

- **[1, 0, 0, 0]**: "ECG";
- **[0, 1, 0, 0]**: "EMG";
- **[0, 0, 1, 0]**: "EDA";
- **[0, 0, 0, 1]**: "RESP";

	One-Hot Encoded label
window 1	[1, 0, 0, 0]
window 2	[1, 0, 0, 0]
window 3	[1, 0, 0, 0]
...	...
window N	[0, 0, 0, 1]

Figure 4.12: Representation of the One-Hot Encoded Label Vector

Chapter 5

Solutions

The problems described in chapter 3 can be reduced to a multiclass classification problem. In a multiclass classification problem there are different data points and different classes. Each data point belongs exclusively to one class. The goal is to create a function that, given a new data point, will be able to correctly predict the class to which the new point belongs. Since the dataset consists of time series, this is a Time Series Classification (TSC) problem.
The steps followed to prepare the data for the ML algorithms are shown in chapter 4. In this chapter I will present three different approaches used in order to experiment different solutions.

Note: All experiments have been carried out on a laptop with an Intel Core i7-7700HQ CPU (4 x 2.8GHz), 16GB DDR4 RAM and a 4GB NVIDIA GeForce GTX 1050Ti GPU.

5.1 1st Approach

After the preprocessing steps common to all the approaches, in this one an additional step involving **Feature Extraction** has been performed in order to extract some significant statistical features. Then, there is the step concerning the creation of the **Statistical Feature Vector** (Fig. 4.9) and the **Plain Labels Vector** (Fig. 4.11). In the end, these vectors are used as train set and are fed to 3 different classifiers: **SVM**, **KNN**, **RF** and a DNN. Furthermore, except for the NN model, the **10-Fold Cross-Validation** has been used as cross-validation technique (Algorithm 1).

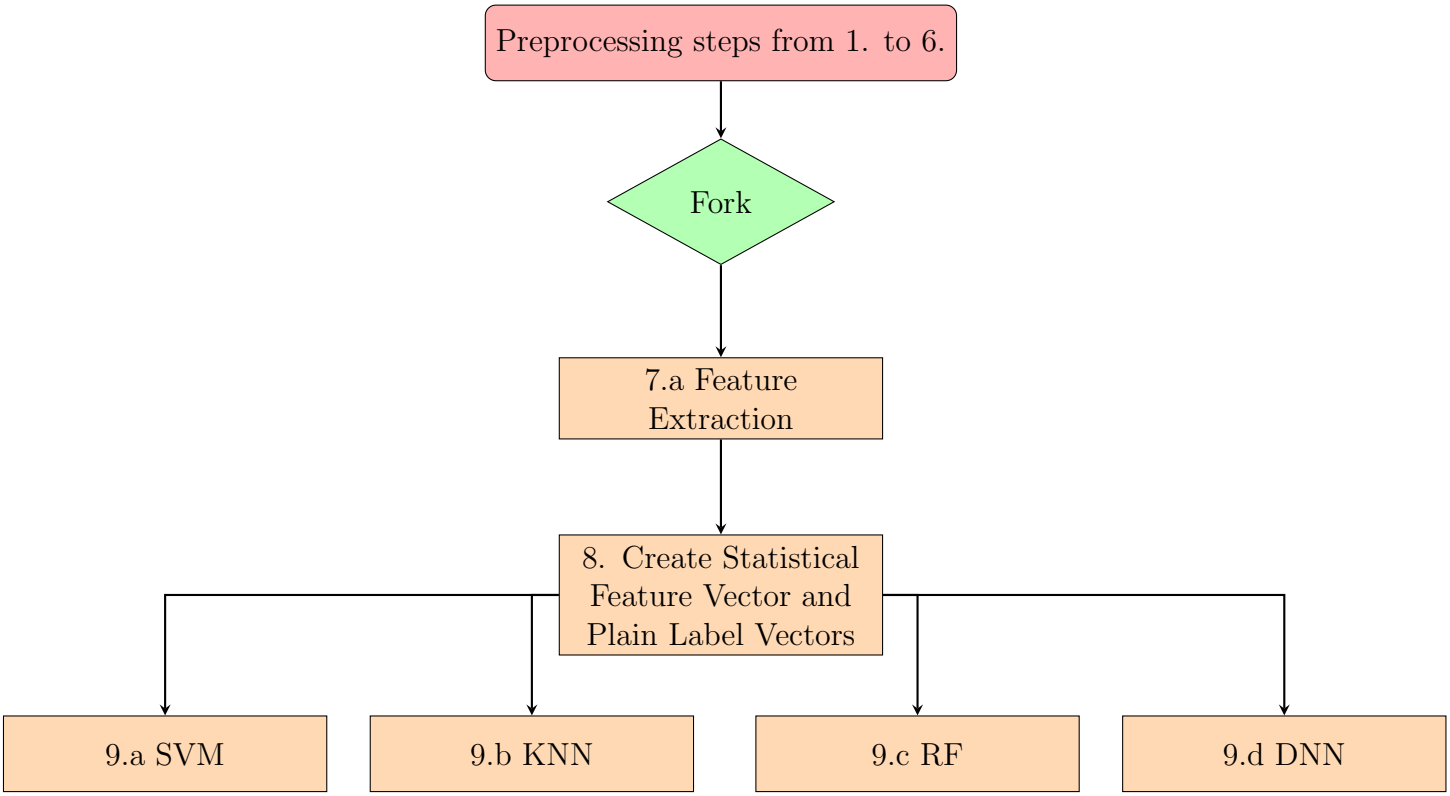


Figure 5.1: Diagram showing the steps of the 1st approach (previous steps in Fig. 4.1).

5.1.1 Support Vector Machines (SVM)

The implementation of the SVM used is the one provided by sklearn library, i.e. `sklearn.svm.SVC` (**Support Vector Classification**). The parameter used are the default ones.
The obtained result with SVM over a 10-Fold CV is **95,84%** mean accuracy with a standard deviation of 0.28%.

5.1.2 K-Nearest Neighbors (KNN)

As highlited in [17], the 1-Nearest Neighbor has very good results for TSC tasks. Indeed, it has been used in this trial. Sklearn's **KNeighborsClassifier** (`sklearn.neighbors.KNeighborsClassifier`) has bean used as KNN implementation. The parameters chosen are **1** for **K** parameter and the default for the remaining ones.
KNN manages to exceed SVM's score by almost 2%, with a **97,66%** mean

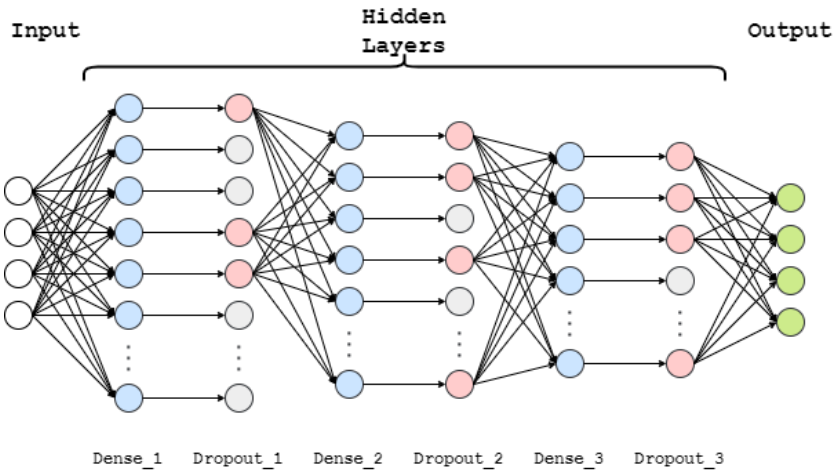


Figure 5.2: Structure of the DNN implemented for the 1st approach

accuracy and a standard deviation of 0.19%.

5.1.3 Random Forest (RF)

As for those above, sklearn's implementation of the **Random Forest** (**RF**) has been used (`sklearn.ensemble.RandomForestClassifier`). The default parameters have been applied except for the **n_estimators** that has been set to **1000**
RF proves to be the most robust model among the 3 tested, outperforming both SVM and KNN with a mean accuracy equal to **98,74%** and a standard deviation of 0,14%.

5.1.4 Dense Neural Network (DNN)

This is the last algorithm used for the 1st approach. The implemented model is a very simple DNN [Fig. 5.2]. Some details about the hyperparameter values are given below. Unspecified hyperparameter values are to be considered as the default ones (to remember that all the NNs implemented in this work use Keras layers).
The input is a 1d array with 4 values, one for each statistical feature extracted. The number of neurons for Dense layers are: 64 (`Dense_1`), 32 (`Dense_2`) and 16 (`Dense_3`). All of them have Relu (Rectified Linear Unit) as activation function. Instead, the dropout rates are: 0.4 (`Dropout_1`), 0.3 (`Dropout_2`) and 0.2 (`Dropout_3`). The `Output` has 4 neurons (one for each class) and Softmax as activation function.

Compiling the model After some trials, the loss function chosen and that performed better for both NNs is Sigmoid Focal Cross Entropy [14] (`tensorflow_addons.losses.SigmoidFocalCrossEntropy`) while Adam has been chosen as optimizer (with a learning rate of 0.0001) and Accuracy as performance metric.

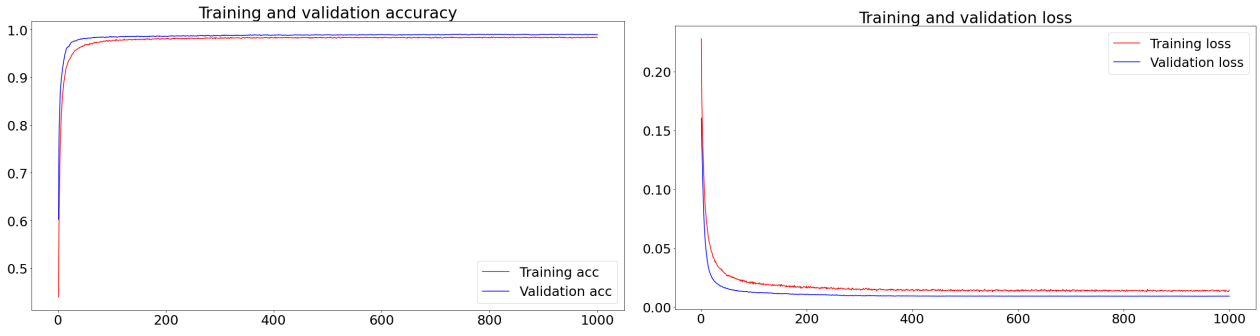
Training and test the model The feature and label vectors have been randomly shuffled and splitted in this way:

- **Train set:** 80%
- **Validation set:** 10%
- **Test set:** 10%

The NNs have been fitted for 1000 epochs, with a batch size of 64 and an early stopping callback (`keras.callbacks.EarlyStopping`) monitoring validation loss with a patience of 100 epochs.
Regarding the hyperparameter values chosen, some are specified while the others are to be considered the default ones of the relative Keras and Tensorflow module. All NN layers used are those implemented by Keras.

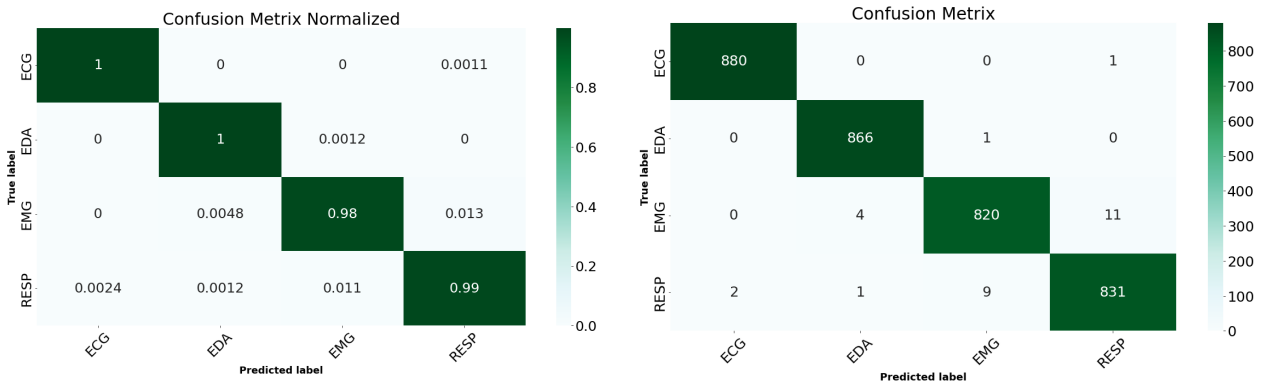
5.1.4.1 Results

The results obtained by the this model are shown in the figures below. The training did not early stopped. The accuracy obtained on the Test set is **99.15%**. Since the accuracy on the Test set is always better than the one on the Training set (the same goes for the loss function), we can conclude that this is an underfit model. Hence, this model could probably be improved by tuning the hyperparameters.



(a) (b)

Figure 5.3: Accuracy (a) and Loss (b) plots of the DNN model (1st approach) on Train (red) and Validation (blue) sets



(a) (b)

Figure 5.4: Confusion matrixes of the DNN model (1st approach) with (a) and without (b) normalization computed on the Test set

5.2 2nd Approach

This experimental path, unlike the first one, does not involve any additional preprocessing step besides the common ones. It uses a **Windows Label Vector** and a **One-Hot Encoded Label Vector** to fed the ML algorightms. The ML algorithms implemented and tested for this path are two NNs: a **Dense NN** and a **1D-Convolutional NN**.

Compiling the model Such as in the DNN of the 1st approach, the loss function that performed better than the others tried is Sigmoid Focal Cross Entropy [14], Adam has been chosen as optimizer (with a starting learning rate of 0.001) and Accuracy as performance metric.

Training and test the model For both NNs, the feature and label vectors have been randomly shuffled and splitted in this way:

- **Train set:** 80%
- **Validation set:** 10%
- **Test set:** 10%

The NNs have been fitted for 1000 epochs, with a batch size of 64, an early stopping callback (`keras.callbacks.EarlyStopping`) monitoring validation loss with a patience of 100 epochs and a learning rate reducer (`keras.callbacks.ReduceLROnPlateau`) that reduce the learning rate by half (until

a minimum of 0.00001) every time that a plateau in the validation loss is encountered within the last 35 epochs, i.e. validation loss is no longer improving.

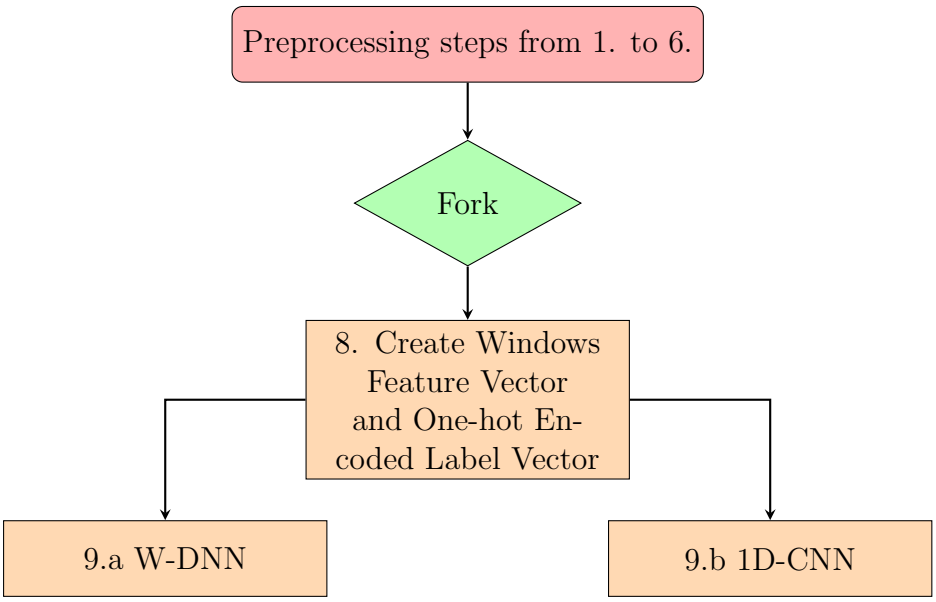


Figure 5.5: Diagram showing the steps of the 2nd approach (previous steps in Fig. 4.1).

5.2.1 Windows Dense Neural Network (W-DNN)

The structure of W-DNN is exactly the same as the one of the DNN of the 1st approach. The only difference is the input size. The input here is a 1D array of 250 elements representing a single window of 2,5 seconds of a signal.

5.2.1.1 Results

The results obtained by the best (i.e. with the highest accuracy) model are shown in the figures below. The training stopped early at 215th epoch. The accuracy obtained on the Test set is **98.80%**

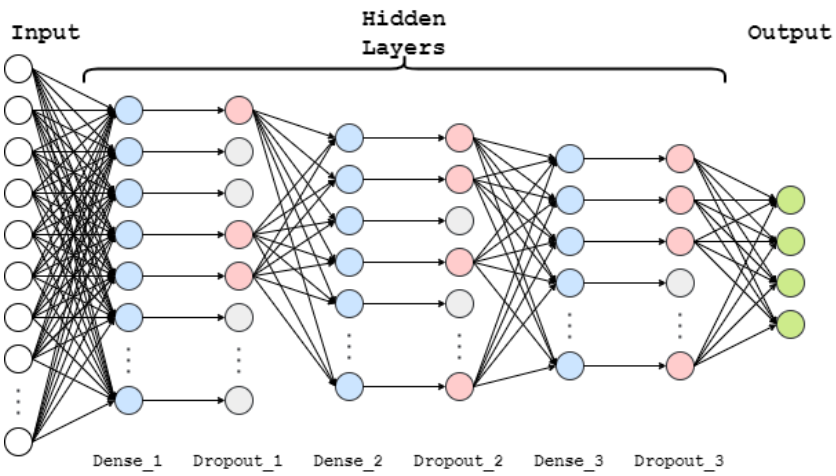
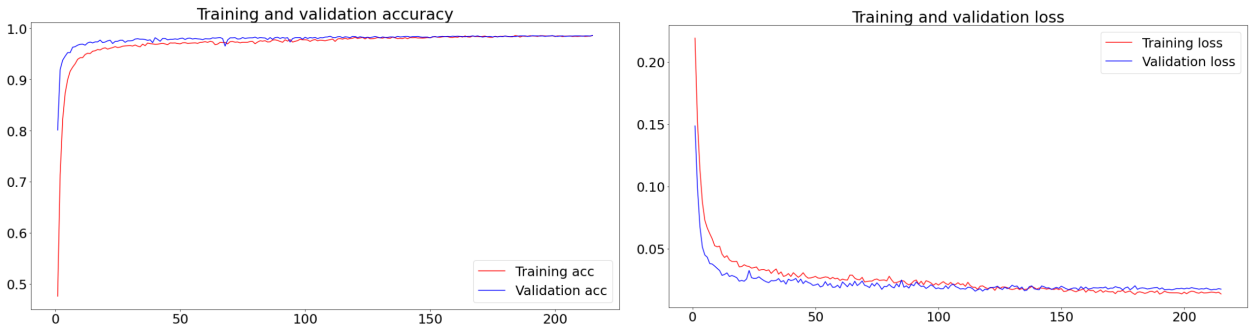
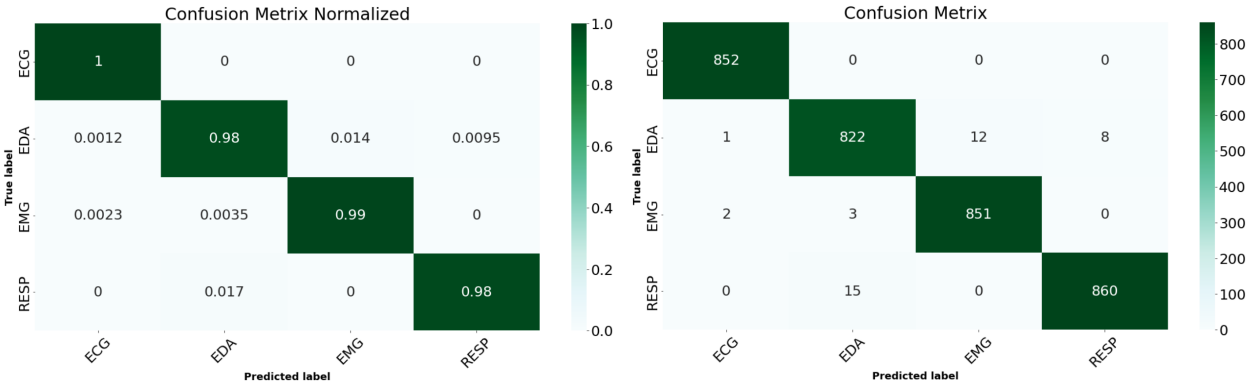


Figure 5.6: Structure of the DNN implemented for the 2nd approach



(a) (b)

Figure 5.7: Accuracy (a) and Loss (b) plots of the DNN model (2nd approach) on Train (red) and Validation (blue) sets



(a) (b)

Figure 5.8: Confusion matrixes of the final DNN model (2nd approach) with (a) and without (b) normalization computed on the Test set

5.2.2 1D Convolutional Neural Network (1D-CNN)

Here is described the second NN approach implemented to solve the classification problem. In this case I opted for a Convolutional NN, precisely a 1D CNN, since the input is a 1D array.
The input is the same as for the DNN showed previously. The Conv layer hyperparameters chosen are: 4 kernel (`Conv1D_1`) and 8 kernel (`Conv1D_2`), both with size 7, Relu as activation function and padding equal to 'same'. Then, a `Global Average Pooling` layer has been inserted to down-sample the data. The `Output` layer is the same as for the DNN, with 4 neurons (one for each class) and Softmax activation function.

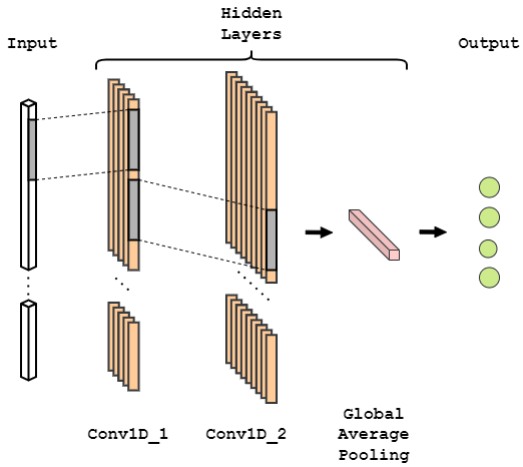
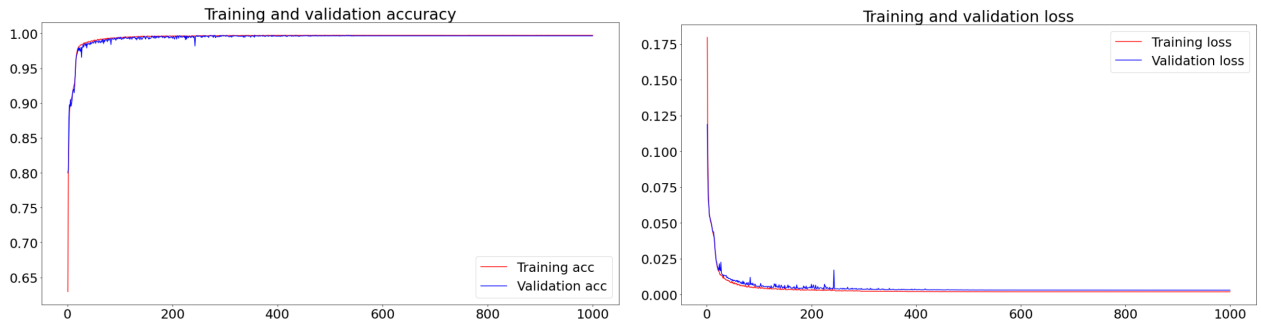


Figure 5.9: Structure of the 1D-CNN implemented for the 2nd approach

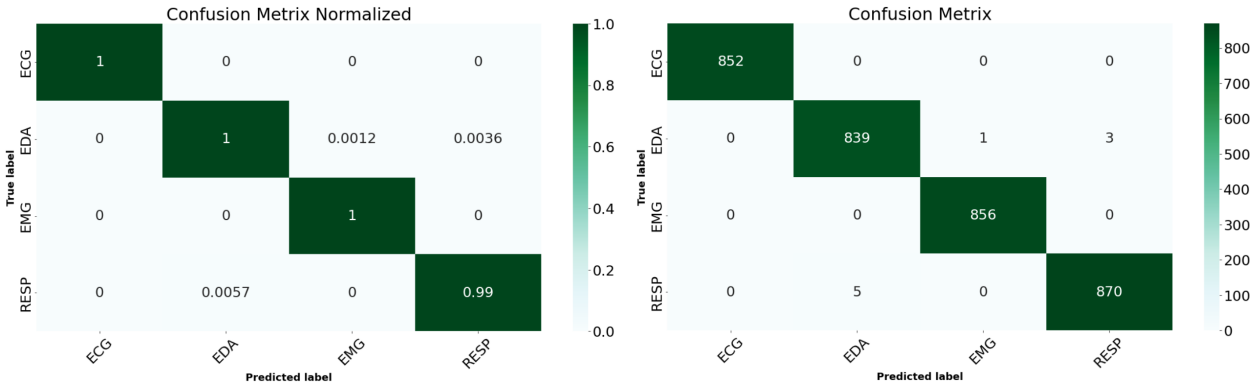
5.2.2.1 Results

The results obtained by 1D-CNN are very impressive. The training proceded for all the 1000 epochs, without early stopping. Hence, it could be trained for more than 1000 epochs to see if it can obtain furher improvements on the accuracy score. The best accuracy obtained on the Test set is **99.73%**.



(a) (b)

Figure 5.10: Accuracy (a) and Loss (b) plots of the 1D-CNN model on Train (red) and Validation (blue) sets



(a) (b)

Figure 5.11: Confusion matrixes of the final 1D-CNN model with (a) and without (b) normalization computed on the Test set

5.3 3rd Approach

Such as the 1st approach, this one adds a further preprocessing step to the common ones, i.e. the **Recurrence Plots generation**, carrying the features from a 1D to a 2D space. There are several works that use rec plots for TSC [10, 12, 9, 11]. In this approach only one NN have been tested. Since we are dealing with images, the choise of the model has fallen on the Convolutional NN, well known for its impressive performances in image classification. The kind of vectors created to train and test the NN are the **Images Feature Vector** (Fig. 4.10) and the **One-Hot Encoded Labels Vector** (Fig. 4.12).

Compiling the model The loss function chosen for both NNs is Categorical Cross Entropy, Adam has been chosen as optimizer (with a starting learning rate of 0.0005) and Accuracy as performance metric.

Training and test the model The feature and label vectors have been randomly shuffled and splitted in the same way as the others NNs (i.e. 80%-10%-10%). The NNs have been fitted for 600 epochs, with a batch size of 64, an early stopping callback monitoring validation loss with a patience of 60 epochs and a learning rate reducer that reduce the learning rate by half (until a minimum of 0.00001) every time that validation loss is not improving within the last 20 epochs.

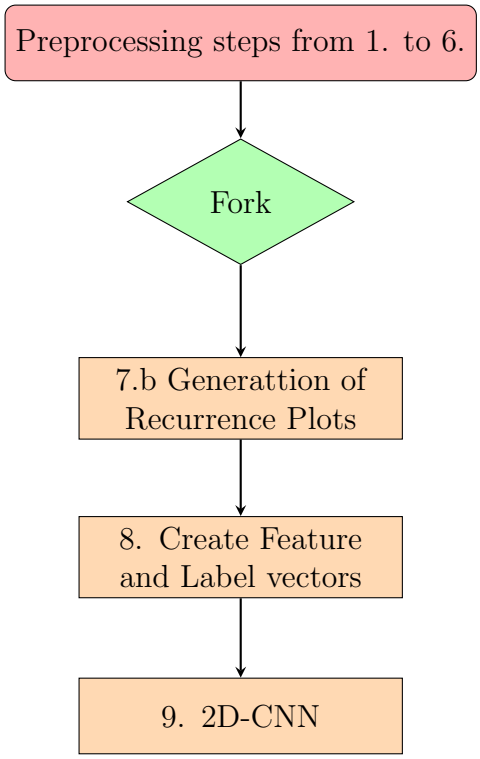


Figure 5.12: Diagram showing the steps of the 3rd approach

5.3.1 2D Convolutional Neural Network (2D-CNN)

The CNN here presented is the deepest network implemented in this work. It has as input the 100x100 image of a signal window, 5 convolutional blocks (`Conv_Pool_Drop`) and a final fully connected block that connects to the Output layer. Each convolutional block is composed of:

1. **Conv layer 2D:** 6 kernels with size 17, Relu as activation function and padding 'same'.
2. **Max Pooling 2D:** kernel size 2.
3. **Dropout:** with a probability of 40% .

The final Fully connected block is composed of a Flatten layer, a Dense Layer with 128 nodes with Relu activation function and a Dropout with a 50% probability. The Output layer is the same as the previuous NNs.

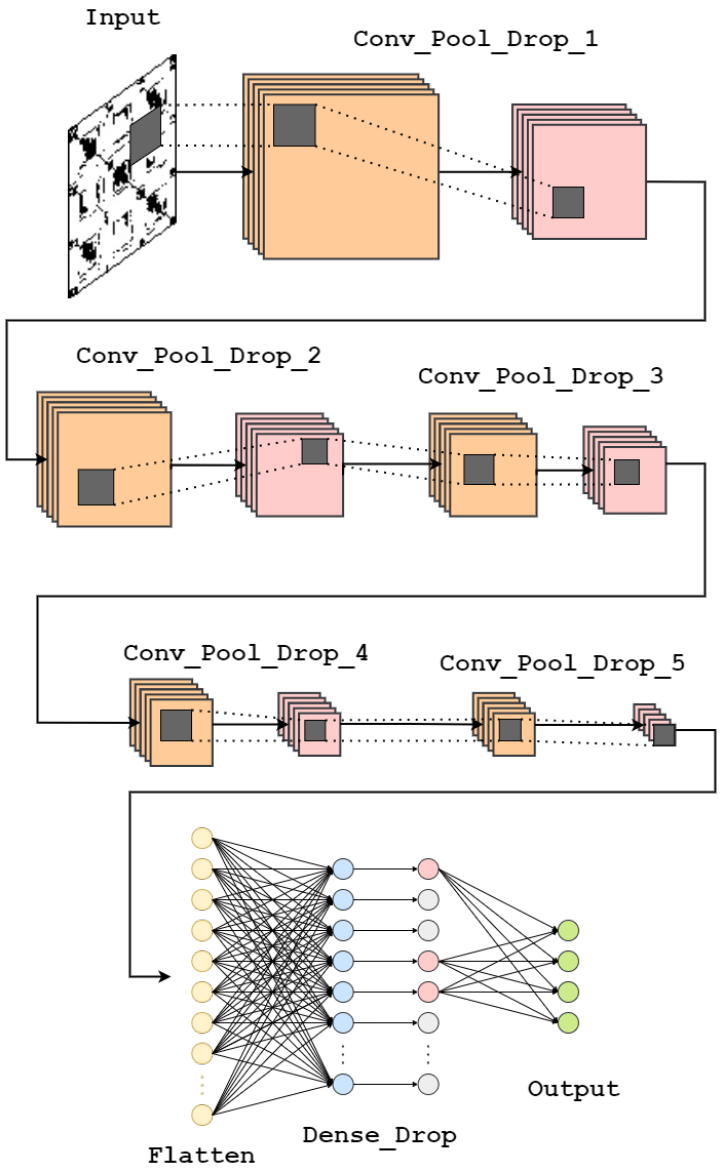
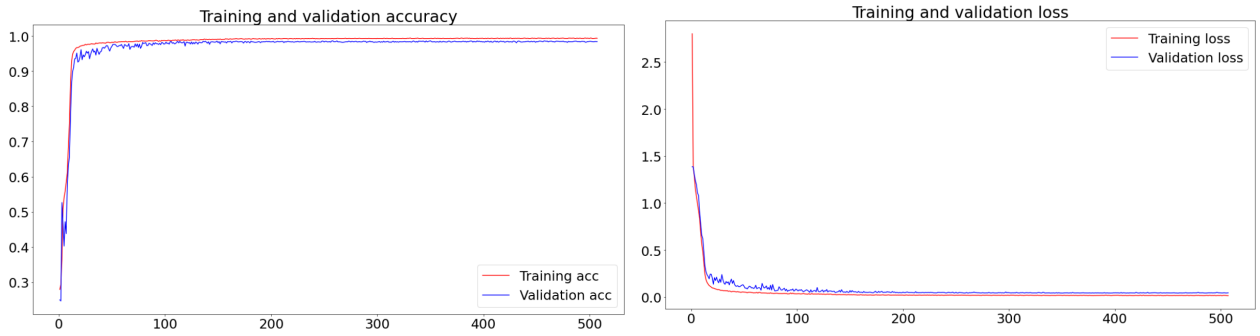


Figure 5.13: Structure of the 2D-CNN implemented for the 3rd approach

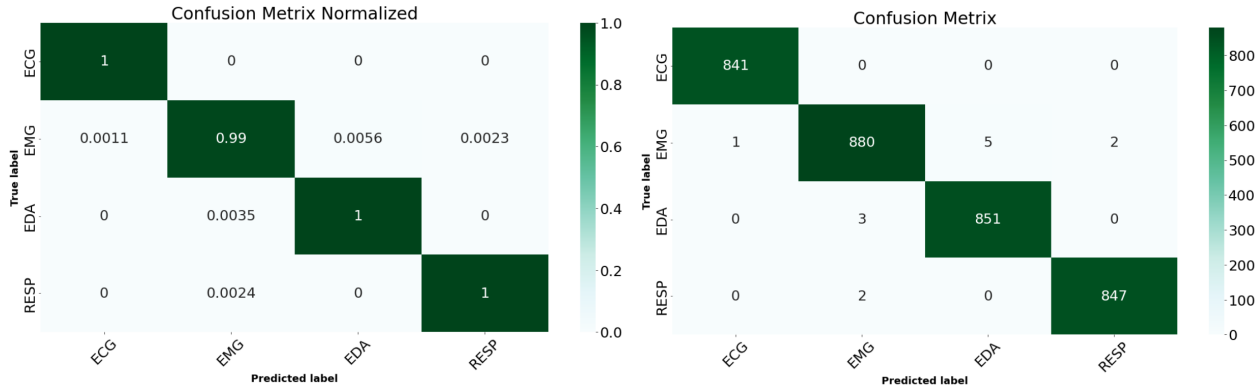
5.3.1.1 Results

The results obtained by 2D-CNN are surely among the best, inferior only to those of the 1D-CNN implemented for the 2nd approach. The training proceded until ealy stopped in the 508th epoch. The best accuracy obtained on the Test set is **99.62%**.



(a) (b)

Figure 5.14: Accuracy (a) and Loss (b) plots of the 2D-CNN model on Train (red) and Validation (blue) sets



(a) (b)

Figure 5.15: Confusion matrixes of the final 2D-CNN model with (a) and without (b) normalization computed on the Test set

5.4 Results comparison

The results obtained with the different approaches tested are very satisfactory. Among all the solutions implemented, the worst accuracy achieved is **95.84%** (scored by SVM in the 1st approach) while the best is **99.74%** (scored by the 1D-CNN in the 2nd approach).
From the resluts obtained it is evident the effectiveness of the NN models that always outperforms SVM, KNN and RF models.

Approach	**Model**	**Accuracy**
1st	**SVM**	95,84%*
1st	**KNN**	97,66%*
1st	**RF**	98,74%*
1st	**DNN**	99,15%
2nd	**W-DNN**	98,80%
2nd	**1D-CNN**	**99,74**%
3rd	**2D-CNN**	99,62%

Table 5.1: Accuracy comparison among the implemented models (accuracies with * are intended as mean accuracies)

Results seems to be very promising, being apparently better than those already obtained by BITalino team in the recent paper [4] (with a best accuracy of approximately **96,7%**). But further investigations are needed to establish it.

Chapter 6

Conclusion and future works

This work aimed to solve a Time Series Classification problem regarding biosignals data, problem encountered by the research center "IT - Instituto de Telecomunicações" of Lisbon in their project BITalino. The dataset used is the WESAD dataset[20] and the signals considered are ECG, EMG, EDA and RESP.
Various Supervised Learning approaches have been implemented and tested, including classic methods such as Support Vector Machines, K-Nearest Neighbors and Random Forest but also various implementation of Neural Networks, testing different variations of both Fully Connected and Convolutional NN models.
The overall results obtained are very promising, showing a tendency of NN models (achieving an accuracy of **99,74%**) to outperform the classical ones (with a best achieved accuracy of **98,74%**) (see Table 5.1).

Many are the improvements that can be brought to this work. One of these is the use of the entire WESAD dataset and other datasets regarding biosignals in order to further validate the results obtained.
Concerning the preprocessing, some filter other than Sivitzky-Golay could be used to denoise the data in a more effective way. A more accurate feature extraction that inlcudes not only mean, standard deviation, skewness and kurtosis could be implemented to reduce dataset to the most significant features coming from different domains. Other more sophisticated windowing functions could be tried out.
Regarding the models, there are plenty of further experiments to be done. K-Fold Cross Validation must surely be implemented for NNs as well. Hyperparameters of the implemented models could be tuned. Many other models and their combinations could to be tested (e.g. LSTM and its variations).

Bibliography

[1] *Recurrence Plots At A Glance.* [http://www.recurrence-plot.tk/glance.php#:~:text=Recurrence%20plot%20%E2%80%93%20A%20recurrence%20plot,a%20certain%20pair%20of%20times].

[2] Anthony Bagnall, Aaron Bostrom, James Large, and Jason Lines. The great time series classification bake off: An experimental evaluation of recently proposed algorithms. extended version, 2016. [https://doi.org/10.1007/s10618-016-0483-9].

[3] Diana Batista, Hugo Plácido da Silva, Ana Fred, Carlos Moreira, Margarida Reis, and Hugo Alexandre Ferreira. Benchmarking of the bitalino biomedical toolkit against an established gold standard. *Healthcare Technology Letters*, 6(2):32–36, 2019.

[4] Patrícia Bota, Ana Fred, João Valente, Chen Wang, and Hugo Plácido da Silva. A dissimilarity-based approach to automatic classification of biosignal modalities. *Applied Soft Computing*, 115:108203, 2022.

[5] Patrícia J. Bota, Chen Wang, Ana L. N. Fred, and Hugo Plácido Da Silva. A review, current challenges, and future possibilities on emotion recognition using machine learning and physiological signals. *IEEE Access*, 7:140990–141020, 2019. [https://doi.org/10.1109/ACCESS.2019.2944001].

[6] Hugo Plácido da Silva, Ana Fred, and Raúl Martins. Biosignals for everyone. *IEEE Pervasive Computing*, 13(4):64–71, 2014.

[7] Maciej Dziezyc, Martin Gjoreski, Przemysław Kazienko, Stanisław Saganowski, and Matjaz Gams. Can we ditch feature engineering? end-to-end deep learning for affect recognition from physiological sensor data. *Sensors*, 20:6535, 11 2020. [https://dx.doi.org/10.3390%2Fs20226535].

[8] Neal B. Gallagher. *Savitzky-Golay Smoothing and Differentiation Filter*, 01 2020. [https://www.researchgate.net/publication/338518012_Savitzky-Golay_Smoothing_and_Differentiation_Filter].

[9] Tameru Hailesilassie. Financial market prediction using recurrence plot and convolutional neural network, 12 2019.

[10] Nima Hatami, Yann Gavet, and Johan Debayle. Classification of time-series images using deep convolutional neural networks, 2017. [https://arxiv.org/pdf/1710.00886.pdf].

[11] Lyudmyla Kirichenko, Tamara Radivilova, Vitalii Bulakh, Petro Zinchenko, and Abed Saif Alghawli. Two approaches to machine learning classification of time series based on recurrence plots. In *2020 IEEE Third International Conference on Data Stream Mining Processing (DSMP)*, pages 84–89, 2020.

[12] Lyudmyla Kirichenko and Petro Zinchenko. *Time Series Classification Based on Visualization of Recurrence Plots*, pages 101–108. 03 2021.

[13] Dawid Laszuk. *recurrence-plot GitHub project*. [https://github.com/laszukdawid/recurrence-plot].

[14] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection, 2018. [https://doi.org/10.1109/ICCV.2017.324].

[15] Michael J. Leonard Michele A. Trovero. *Time Series Feature Extraction*, 2018. [https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/2020-2018.pdf].

[16] Kizito Nkurikiyeyezu, Anna Yokokubo, and Guillaume Lopez. The effect of person-specific biometrics in improving generic stress predictive models, 2019. [https://arxiv.org/abs/1910.01770v2].

[17] Tiago Santos and Roman Kern. *A Literature Survey of Early Time Series Classification and Deep Learning*, 2016. [http://ceur-ws.org/Vol-1793/paper4.pdf].

[18] Abraham M. J. E. Savitzky Golay. *Smoothing and Differentiation of Data by Simplified Least Squares Procedures*, 1964. Anal. Chem. 1964, 36, 8, 1627–1639, [https://doi.org/10.1021/ac60214a047].

[19] Philip Schmidt, Attila Reiss, Robert Dürichen, and Kristof Van Laerhoven. Wearable-based affect recognition—a review. *Sensors*, 19(19), 2019. [https://doi.org/10.3390/s19194079].

[20] Philip Schmidt, Attila Reiss, Robert Duerichen, Claus Marberger, and Kristof Van Laerhoven. Introducing wesad, a multimodal dataset for wearable stress and affect detection. In *Proceedings of the 20th ACM International Conference on Multimodal Interaction*, ICMI '18, page 400–408, New York, NY, USA, 2018. Association for Computing Machinery. [https://doi.org/10.1145/3242969.3242985] [https://archive.ics.uci.edu/ml/datasets/WESAD+%28Wearable+Stress+and+Affect+Detection%29].